ARL-TN-0674 ● May 2015

US Army Research Laboratory

Cost Computations for Cyber Fighter Associate

by Andrew Erbs and Lisa M Marvel

ARL-TN-0674 ● May 2015

US Army Research Laboratory

Cost Computations for Cyber Fighter Associate

by Andrew Erbs
Cyber Security Collaborative Research Alliance Student, Pennsylvania State University, State College

Lisa M Marvel
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* May 2015	**2. REPORT TYPE** Final	**3. DATES COVERED (From - To)** 1 June 2014–31 January 2015
4. TITLE AND SUBTITLE Cost Computations for Cyber Fighter Associate		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Andrew Erbs and Lisa M Marvel		**5d. PROJECT NUMBER**
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Research Laboratory ATTN: RDRL-CIN-D Aberdeen Proving Ground, MD 21005		**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TN-0674
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		**10. SPONSOR/MONITOR'S ACRONYM(S)**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Modeling cost in cyber operations is a difficult problem due to the number of variables involved and the potential perspectives that can be used to evaluate what constitutes cost. As such, there is a need for a cost computation method and program for the Cyber Fighter Associate (a knowledge-based system that helps to evaluate agility maneuvers). We developed a cost computation method and program for evaluating these costs from a number of perspectives while allowing for the changing network topology of real-world battlefield situations. This cost computation method allows for quick computation of cost for a given course of action and the potential for the addition of different approaches to cost analysis with a minimum of effort.

15. SUBJECT TERMS

cyber security, software patch management, tactical networks, cyber modeling, cyber simulations

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Lisa M Marvel
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	58	**19b. TELEPHONE NUMBER (Include area code)** 410-278-6508

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

Preface

This work was developed during a summer internship between my junior and senior year in Computer Engineering at Pennsylvania State University. I (Andrew Erbs) inquired with Dr Patrick McDaniel during my junior year about potentially becoming a research assistant with the Systems and Internet Infrastructure Security (SIIS) laboratory at Pennsylvania State University. He agreed, and I began work with the lab the next semester. One part of SIIS's ongoing research efforts includes working in collaboration with the Cyber-Security (CSec) Collaborative Research Alliance (CRA). The CSec CRA is sponsored by the US Army Research Laboratory (ARL), and I was given an opportunity to conduct research over the summer with the Computational and Information Sciences Directorate. Thus, this research is part of a larger ongoing body of research directed toward the goals of the CSec CRA.

Additionally, this report documents a portion of a larger project. There are 2 related works that were done in concert. The first is the *Cyber Fighter Associate*, which is currently in press with (ARL).[1] The second is a *Communication Protocol for CyAMS and the Cyber Fighter Associate Interface*, also documented in an ARL technical note, ARL-TN-0673.[2]

[1] Huber C, Marvel LM. Cyber fighter associate. Aberdeen Proving Ground (MD): Army Research Laboratory (US); in press.

[2] Harman D, Brown S, Henz B, Marvel LM. A communication protocol for CyAMS and the Cyber Fighter Associate Interface. Aberdeen Proving Ground (MD); Army Research Laboratory (US); 2015 May. Report No. ARL-TN-0673.

INTENTIONALLY LEFT BLANK.

1. Introduction

The ability to provide perfectly secured digital information seems like an impossible task. With the advent of each new security measure, academics and malicious agents spend large amounts of time and effort to subvert the gains achieved. The goals of these academics and malicious agents are quite disparate, but both end up with the same result: new security measures subverted. For this reason, the network administrator's job is a difficult one, and because no connected network is perfectly secure, the administrator has to focus on damage mitigation. However, damage mitigation is not an easy task because the administrator must make a wide variety of decisions in a rapid succession. The Cyber Fighter Associate (CyFiA) may be a very valuable tool to implement under these conditions.

The CyFiA is a program currently being developed by Charles Huber.[1] The program is designed to take in vast amounts of information from the status of the network and present options to network administrators. These options are based on the goals of the network administrator and ongoing cyber operations. The CyFiA originates from a similar program called the Warfighter Associate, which assists with on-the-ground decision making during missions. Because work on the CyFiA is just beginning, efforts have been focused on one particular scenario: maintaining capability on a critical path in a network using a variety of agility methods.

Ongoing work is being developed to increase the number of scenarios on which CyFiA can provide guidance. Because the CyFiA is being developed with potential scaling in mind, tests occur with an NS-3-based program called CyAMS that, alongside a high-computing resource, allows testing to include scenarios involving millions of nodes. Because of this large-scale size, cost computations of agility actions can be highly intensive, especially when data transmission paths must be created from a single source.

As a result of these new strategies, a distinct entity to provide cost analysis for the CyFiA is needed. A cost analysis program would allow the CyFiA to focus solely on decision making and utilize levels of abstraction concerning the cost of agility maneuvers. As a result, more capability can constantly be added to the cost computations without a reduction in capability for the decision-making apparatus.

2. Design

The design of the cost computation apparatus has a few key design components that greatly shaped the overall structure of the program. As described in Section 1, because computational requirements may be quite varied on the different components of the CyFiA, the various components must be able to communicate over a network. In addition, because cost computations are handled independently from other components of the CyFiA, the cost computation program should possess an ability to represent the network in its entirety.

However, there is a large quantity of information that must be passed over the network as a result of this scheme. Updates to a given node or edge in the graph must be passed into the onboard representation of the network. If they are not, the cost computations will be incorrect. Thus, there is a large amount of network traffic, specifically status updates, and all must be processed before a given cost analysis can be performed. Therefore, the program required communication over a number of ports to transfer this information in a timelier manner.

Figure 1 depicts the current model for communications between the various components of the CyFiA. Of the 7 ports used to listen for incoming data, 6 of them are used to track status change information coming from the graph. The last port is the request port, and the other components of the CyFiA request cost computations or graph path requests over this port. Because a received packet on any of these ports requires an action from the cost program, multithreaded listeners must be used. Details of the communication protocol developed to enable commication between CyFiA, the cost computation program described herein and CYAMS can be find in Harman, et al.[2]

A specific class called ListenThread was created for multithreaded listeners. When ListenThread is instantiated, it is passed a given port on which to listen. Once it receives information, it appends the port number it is listening on to the received string. Once this process has been completed, it pushes the received string to a blocking priority queue, and depending on the port on which it was received, it sorts these strings into status changes, graphical user interface (GUI) requests, and computation requests. This process allows the cost computation program to handle the given updates and requests in the correct order to avoid a majority of race conditions that may occur due to incorrect status information in the onboard graph representation.

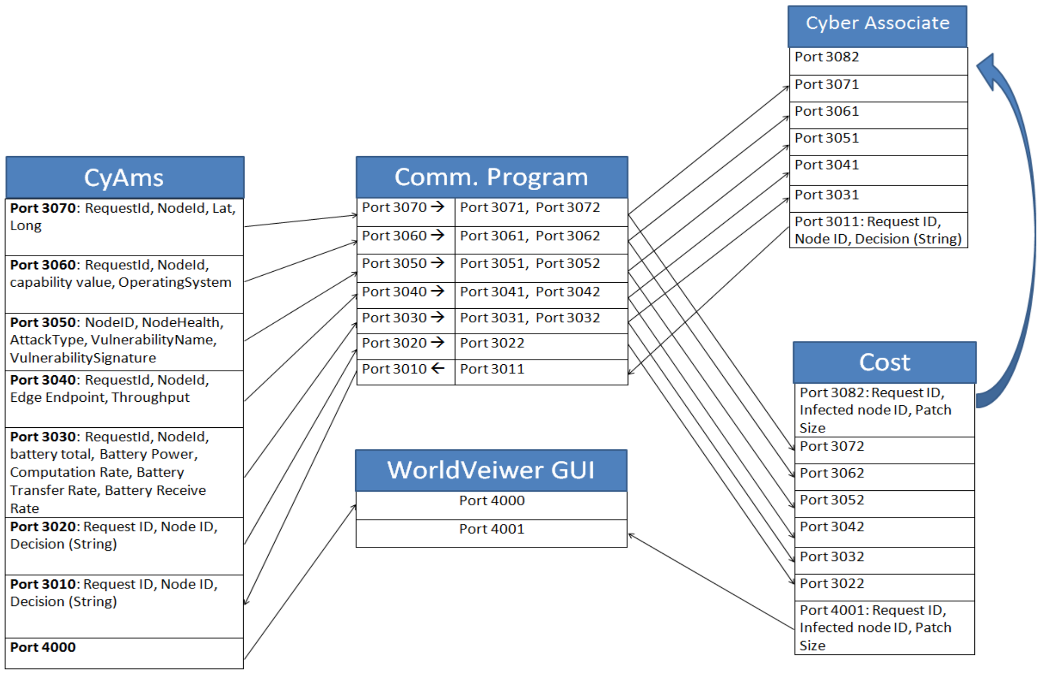
CyAms
Port 3070: RequestId, NodeId, Lat, Long
Port 3060: RequestId, NodeId, capability value, OperatingSystem
Port 3050: NodeID, NodeHealth, AttackType, VulnerabilityName, VulnerabilitySignature
Port 3040: RequestId, NodeId, Edge Endpoint, Throughput
Port 3030: RequestId, NodeId, battery total, Battery Power, Computation Rate, Battery Transfer Rate, Battery Receive Rate
Port 3020: Request ID, Node ID, Decision (String)
Port 3010: Request ID, Node ID, Decision (String)
Port 4000
Comm. Program
Port 3070 → Port 3071, Port 3072
Port 3060 → Port 3061, Port 3062
Port 3050 → Port 3051, Port 3052
Port 3040 → Port 3041, Port 3042
Port 3030 → Port 3031, Port 3032
Port 3020 → Port 3022
Port 3010 ← Port 3011
WorldVeiwer GUI
Port 4000
Port 4001
Cyber Associate
Port 3082
Port 3071
Port 3061
Port 3051
Port 3041
Port 3031
Port 3011: Request ID, Node ID, Decision (String)
Cost
Port 3082: Request ID, Infected node ID, Patch Size
Port 3072
Port 3062
Port 3052
Port 3042
Port 3032
Port 3022
Port 4001: Request ID, Infected node ID, Patch Size

Fig. 1 Model showing the port protocol between the programs

The parent in the program handles each string in the queue. The handler must be single-threaded otherwise status updates would contain race conditions with the cost analysis requests. The handler executes in a continuous loop after creating its children until it receives an administrative string on a specific port that tells the program to end or print the current representation of the graph. Because a blocking priority queue is used for the interaction between the listeners (children) and the handler (parent), the handler will always process waiting status updates before attempting any cost analysis. The implicit assumption is that at some point, no status updates will be in the queue, and at that time the cost program will handle any cost computations.

The creation and termination of the cost computation program will eventually be handled by an external entity, and when this occurs, the communication for the termination of the program must be encrypted so that an adversary or malicious attacker cannot compromise the integrity of the system by simply telling the cost program to terminate. Figure 2 illustrates the simplified flow chart of the cost analysis program. Of critical interest is the order of the yes or no blocks. The higher ones will be processed first and return to the Input Received block before the lower ones are processed. The code listing for the cost computation program can be found in the Appendix.

Program Flow

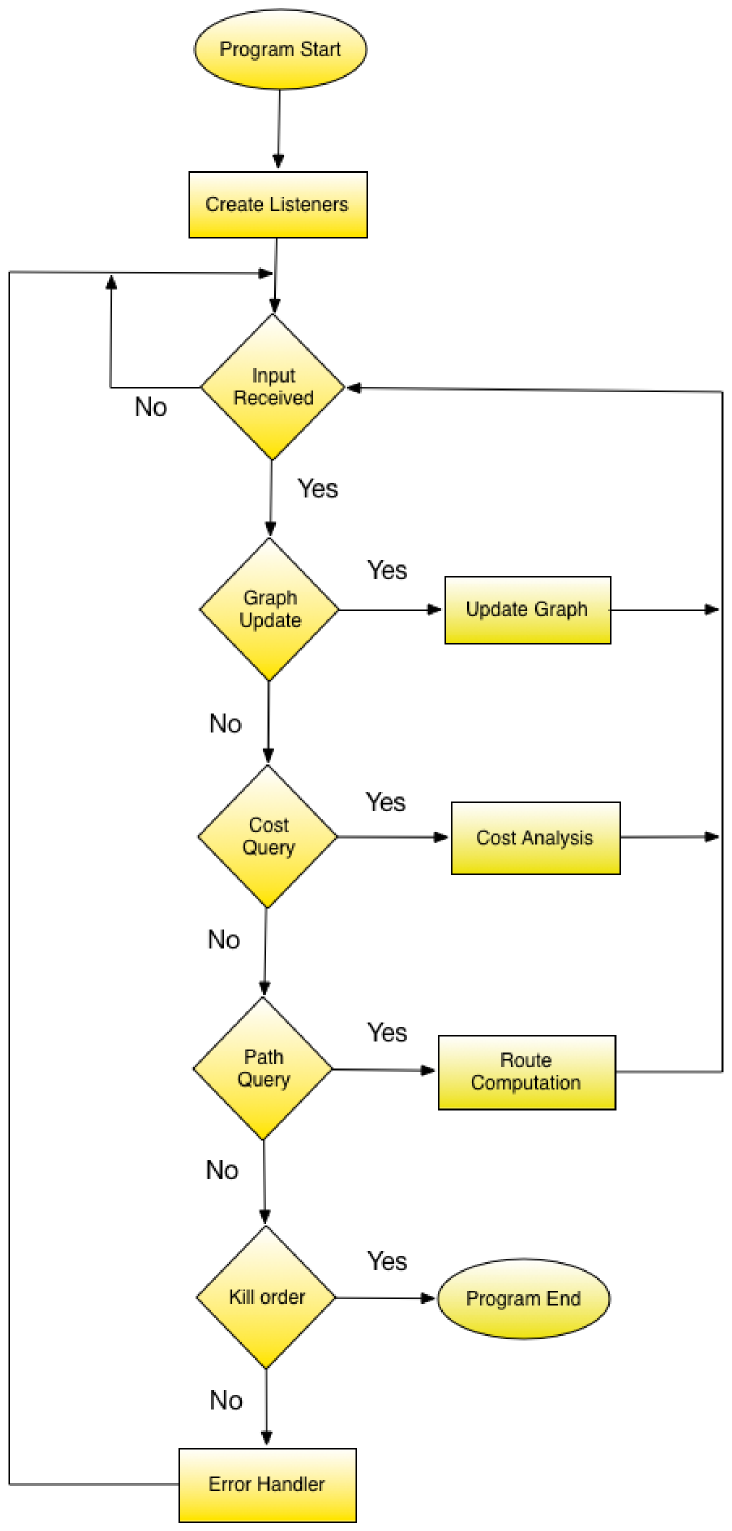

Fig. 2 Flow chart depicting the simplified flow of the cost computation program

The next critical areas of the program are the onboard graph representation and how all of the status updates are handled. Because there are no guarantees that vertexes and edges cannot be added during a scenario, each piece of graph information is checked to determine whether the target vertex or edge is already in the graph. Once this is done, the request is passed to 2 different handlers, depending on whether the target vertex or edge was in the graph.

If the target vertex was not already present in the graph, then a specific constructor is used to create the vertex with the given values and default values for all other information that is required. In the case of edge information, 2 additional questions are necessary: 1) whether the end vertex of the edge is present in the graph and 2) whether this specific edge already exists between these 2 vertexes. If the end vertex is not present, then by definition that edge could not already exist, and as such the end vertex and the edge are created with the information given. If the edge is not present but the 2 vertices are, then the edge is simply created. Finally, if the edge or vertex is already present, it is simply updated with the new information.

It is clear from Fig. 1 that a large amount of specific information is being passed. Some particular pieces of information such as geographic location and operating system (OS) are not currently used, although they are updated in the onboard representation of the graph. These are pieces of information that will be factored into the cost analysis in future work, and in this way the cost computation has a large amount of maturation still possible.

The other critical area of the program is the computation request handler, which also handles critical path requests. As mentioned previously, the critical path is the “best route” between the 2 important nodes in the simulation. In the GUI, the user will select different origin and endpoint nodes and then the critical path will be selected between them based on speed of transit of data. This path will then be protected by the CyFiA by employing various agility maneuvers. This request and other future requests to reroute the critical path based on the infection state of nodes are handled in the computation request portion of the program. However, it is relevant because the GUI request will only happen once during a simulation, these requests are ranked above cost requests from the CyFiA in the priority queue.

There are several potential cost analysis functions that the cost program can do currently: 1) it can calculate the cost to patch a given node with a specific patch of a specific size; 2) it can calculate the cost to block a specific adjacent node-to-node connection in the graph; 3) it can provide the cost to quarantine a specific function of a program running on the node that is vulnerable; and 4) it can provide the cost to have a particular node heal itself from a vulnerability. For each of these possible cost computations, a different handler is called. The handlers for the quarantine, blocking, and healing are all calculated offline using similar platforms as the node in question. They also send the loss in capability from the quarantine, the lost throughput on the given edge, and the loss in capability from healing, respectively, back to the CyFiA.

The handler for the cost analysis of a patch operation is substantially more complex than these other 3 handlers. The cost of a patch operation can be measured in 2 principal ways: battery cost and time to completion. Because it is impossible to relate these 2 numbers into a single cost analysis, the CyFiA asks for a specific cost analysis in one of these 2 categories. If the time to completion computation is requested, then the best path between the 2 nodes is computed using Djikstra's algorithm for best path. Once this process is completed, the cost to traverse this path is computed by taking the patch size and multiplying it by the summation of the throughput on each edge that is traversed along the best path. The result is the amount of time (in seconds) required to transfer the patch. The result is then sent back to the CyFiA.

If the battery impact computation is requested, then the best path is computed using Djikstra's algorithm. The cost to traverse this path is then computed by taking the patch size and multiplying it by the summation of the battery weights on each edge that is traversed along the best path. The battery weight on each edge is the summation of the battery cost to send from the sending node along with the battery cost to receive from the receiving node. This results in a battery impact in watt hours. The result is then sent back to the CyFiA.

This method comprises all of the functionality available at the current time in the cost computation program for the CyFiA. Figure 3 illustrates the processes described above.

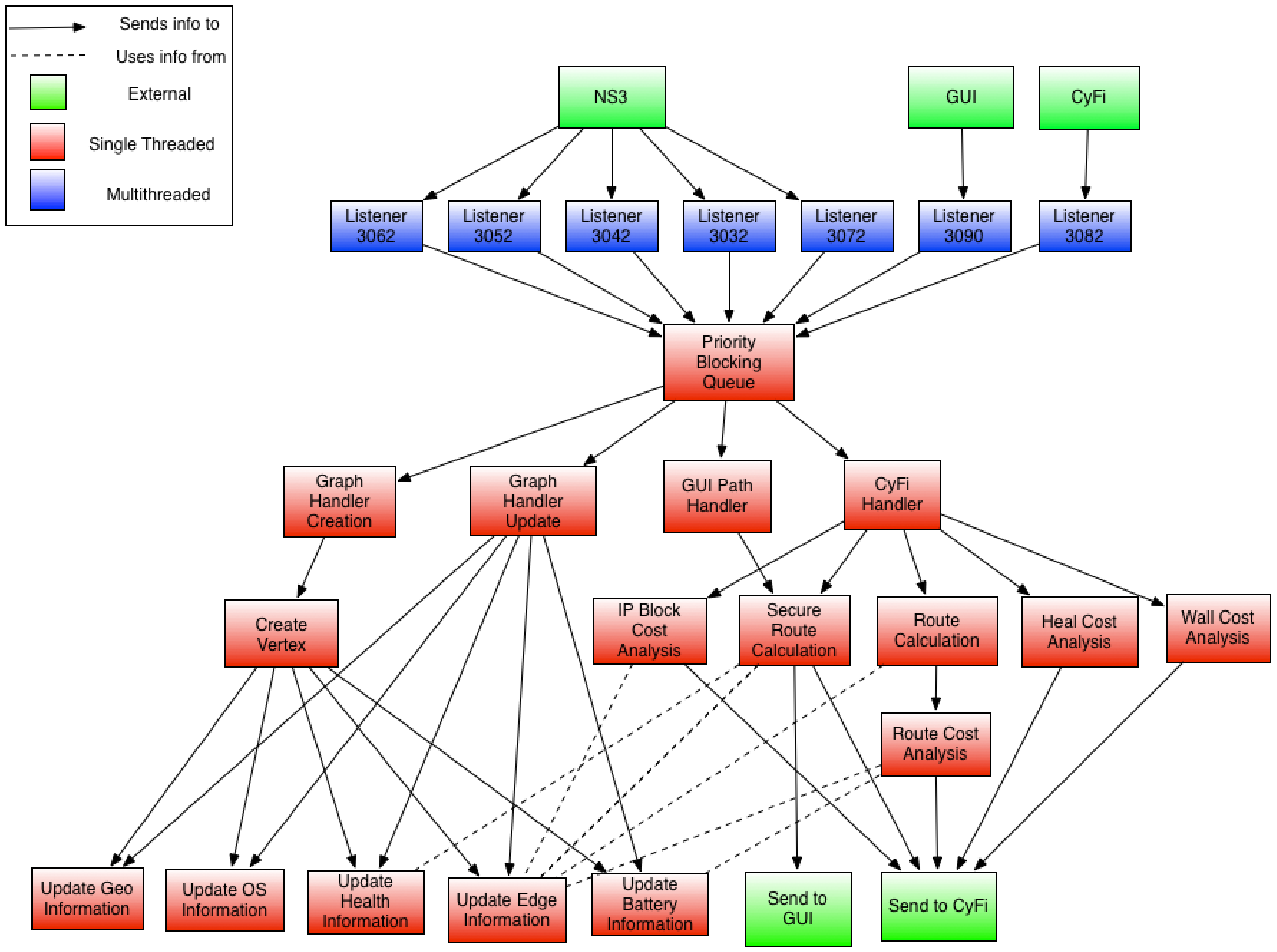

Fig. 3 Detailed program graphic depicting the composition of the cost computation program

3. Conclusions and Future Work

Modeling cost of agility maneuvers in cyber operations is very difficult and not straightforward. As such, it was important to create a tool that could be implemented to do cost computations for CyFiA. The previously stated methodology does this in a standalone package, and as such it is versatile and it can potentially be used in a number of different circumstances with minor changes. Overall, the cost computation program solves critical problem, but it has the potential to provide a significant amount of additional functionality. This would be the subject of future work.

Future studies should include work in 3 areas. The first area to pursue is a real-time updating graph that shows 2 things: 1) the summation of the health of all the nodes versus time and 2) the battery level summation of all of the nodes versus time. These graphs, alongside a metric of "time to 100% operational", could be a powerful tool that might facilitate some very interesting research on priorities in network security and their results on the state of the network. The second area to pursue is a more efficient computation of best path. Because decisions in network security must be made rapidly at times, a faster computation of best path could be helpful . To accomplish this goal, algorithms with heuristics seem to be the best choice; however, a heuristic process needs to be created for traversing a network before this path can be used. The last area to pursue is adding safeguards into the throughput patch plan and the battery patch plan. These safeguards could be used to check whether the best plan will expend all of the battery available, and if so, it could recommend another plan. This does necessitate a larger question of establishing the lowest acceptable battery power.

In conclusion, there is always more depth that can be added to the cost computations given, but when should that depth arrive in more precise forms of query, and when should that precision originate from an abstracted "computation" are questions that must be resolved.

4. References

1. Huber C, Marvel LM. Cyber fighter associate. Aberdeen Proving Ground (MD): US Army Research Laboratory (US); in press.

2. Harman D, Brown S, Henz B, Marvel LM. A communication protocol for CyAMS and the cyber associate interface. Aberdeen Proving Ground (MD): US Army Research Laboratory (US); 2015. Report No.: ARL-TN-0673.

INTENTIONALLY LEFT BLANK.

Appendix. Code Listing

This appendix appears in its original form, without editorial change.

```
//written by Andrew Erbs

/*
   unit for throughput is in megabytes per second

 *POTENTIAL* factor in loss of battery power
 *POTENTIAL* factor in computational battery cost with
suseptibility

*/

/*
   order to work on:

   10)  resiliency in the network (time to operational)
   A) charts
   14) figure out undefined behavior if it cant get to
endpoint(i think it stays infinity)
   in general test program
   */

import java.net.*;
import java.util.*;
import java.io.*;
import java.util.concurrent.*;

//this is the overall class structure which will
//implement all of the cost/risk code
public class davidCost {

      static BlockingQueue<String> workToDo = new
ArrayBlockingQueue<String>(1024);
      static List<Vertex> Graph;
      static InetAddress localaddr;

      public static void main (String[] args) {

           try {//tries out the networking code for a local
address
                localaddr =
InetAddress.getByName("127.0.0.1");
           }
           catch( UnknownHostException e) {
                System.out.println("Couldn't find local
address");
           }

           ArrayList<Thread> tList = StartListeners();


           Graph = new ArrayList<Vertex>();
```

```
          Thread t = new Thread(new Runnable(){

                    @Override
                    public void run() {
                    boolean programEnd = false;
                    while(!programEnd) {

                    String line= "";
                    String[] lineInfo;
                    try {
                    line = workToDo.take(); //will block
until it recieves a message
                    }
                    catch (InterruptedException e) {
                    System.out.println("the parent take
operation was interrupted");
                    }
                    catch (Exception e) {
                    System.out.println("the parent take
operation threw unknown exception");
                    //programEnd=true;//this time it is
arbitrary end to the input loop ************why is this
taken out?
                    }
                    lineInfo=line.split(",");
                    boolean parses=true;
                    for(int ind = 0; ind < 3; ind++) {
                         try {
                              int temp =
Integer.parseInt(lineInfo[ind]); //*************how is
there only one integer
                         }
                         catch (NumberFormatException e)
{

     System.out.println("Argument " + ind + " does not
parse to an integer");
                              parses=false;
                         }
                    }
                    if(lineInfo.length < 3) {
                         System.out.println("there was
an error parsing the operation: length is less than 3");
                         parses=false;
                    }
                    else if (parses &&
Integer.parseInt(lineInfo[0])==3082) {//if it did originate
from chuck's port
                         try {
                              int temp =
Integer.parseInt(lineInfo[3]);
                         }
```

```
                            catch (NumberFormatException e)
{
                                System.out.println("Patch
Size does not parse to an integer in request" +
lineInfo[1]);
                                parses=false;
                            }
                            if(parses) {

    System.out.println("handleing request " );
                                for(int i = 0 ; i <
lineInfo.length; i++){

    System.out.println("reequst " + lineInfo[i]);
                                }
                                HandleRequest(lineInfo);
                                PrintGraph();
                                programEnd=true;//this
time it is arbitrary end to the input loop
                            }
                        }
                        else if (parses &&
Integer.parseInt(lineInfo[0])==3022) {//if it did originate
from scott's port
                            //execute the critical path
computation
                            HandleGuiPath(lineInfo);
                        }
                        else if (parses &&
Integer.parseInt(lineInfo[0])==3002) {//if it originate
from kill port
                            int argument =
Integer.parseInt(lineInfo[1]);
                            if(argument==0) {
                                PrintGraph();
                            }
                            else {
                                programEnd=true;

    System.out.println("Received kill order, committing
seppuku...");
                            }
                        }
                        else if (parses) {
                            int location =
SearchForNode(Integer.parseInt(lineInfo[2]));
                            if(location!=-1) {//is in the
graph
                                HandleUpdate(lineInfo,
location);
                            }
                            else {//is not in the graph
```

```
                                HandleCreation(lineInfo);
                        }
                    }
                }
                }

            });

            tList.add(t);

            for(int i = 0; i < tList.size(); i++){
                tList.get(i).start();
            }
            for(int i = 0; i < tList.size(); i++){
                try {
                    tList.get(i).join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

        }
        ///////////////////////////////////////////////////////
/////////////////////////////
        /*
Function:    StartListeners
Arguments:
Explanation: Creates all of the listeners on the various
ports that are used
Returns:
*/
        ///////////////////////////////////////////////////////
/////////////////////////////
        private static ArrayList<Thread> StartListeners() {
            ArrayList<Thread> tList = new
ArrayList<Thread>();

            //creates the threads to listen
            ListenThread geoThread = new ListenThread(3072);
            Thread geoThreadHead = new
Thread(geoThread);//thread based on an instance of a class
            tList.add(geoThreadHead);
            //geoThreadHead.start();//runs the run method
            //port, requestID, NodeID, Lat, Long


            ListenThread capThread = new ListenThread(3062);
            Thread capThreadHead = new
Thread(capThread);//thread based on an instance of a class
            tList.add(capThreadHead);
            //capThreadHead.start();//runs the run method
            //port, requestID, NodeID, Operating System
```

```
           ListenThread healthThread = new
ListenThread(3052);
           Thread healthThreadHead = new
Thread(healthThread);//thread based on an instance of a
class
           tList.add(healthThreadHead);
           // healthThreadHead.start();//runs the run
method
           //port, RequestID, NodeID, NodeHealth,
AttackType, VulnerabilityName, VulnerabilitySignature

           ListenThread edgeThread = new
ListenThread(3042);
           Thread edgeThreadHead = new
Thread(edgeThread);//thread based on an instance of a class
           tList.add(edgeThreadHead);
           //edgeThreadHead.start();//runs the run method
           //port, requestID, NodeID, Edge Endpoint,
Throughput

           ListenThread batteryThread = new
ListenThread(3032);
           Thread batteryThreadHead = new
Thread(batteryThread);//thread based on an instance of a
class
           tList.add(batteryThreadHead);
           // batteryThreadHead.start();//runs the run
method
           //port, requestID, NodeID, battery total,
Battery Power, Computation Rate, Transfer Rate, Recieve
Rate

           ////////////////////////////////////This is the
thread for listening to chuck///////////////

           ListenThread requestThread = new
ListenThread(3082);//3081 for sending to chuck
           Thread requestThreadHead = new
Thread(requestThread);//thread based on an instance of a
class
           tList.add(requestThreadHead);
           // requestThreadHead.start();//runs the run
method
           //port, requestID,startNode, endNode, plan #,
patchsize (should be 0 in all plans but "4")

           ////////////////////////////////////This is the
thread for listening to scott///////////////

           ListenThread criticalThread = new
ListenThread(3022);//3022 for sending to scott
```

```
		Thread criticalThreadHead = new
Thread(criticalThread);//thread based on an instance of a
class
		tList.add(criticalThreadHead);
		// criticalThreadHead.start();//runs the run
method
		//port, reqestID, origin, endpoint

		////////////////////////////////This is the
thread for ending the program

		ListenThread endThread = new
ListenThread(3002);//3022 for sending to scott
		Thread endThreadHead = new
Thread(endThread);//thread based on an instance of a class
		tList.add(endThreadHead);
		// endThreadHead.start();//runs the run method
		//port, reqestID, origin, endpoint
		return tList;
	}
	/////////////////////////////////////////////////////
/////////////////////////////
	/*
Function:    HandleCreation
Arguments:   String [] line
Explanation: Takes in the array of string arguments
received over the network,
and depending on the port it was received on, creates the
node
with the given infotmation. If edge data is received, it
will
create end node if it doesn't exist. Default values are
used when none are specified.
Returns:
*/
	/////////////////////////////////////////////////////
/////////////////////////////
	private static void HandleCreation (String [] line) {
		if(line[0].compareTo("" + 3072)==0) {//this is
the geo thread
			//port, requestID, NodeID, Lat, Long
			HandleCreationGeo(line);
		}
		else if(line[0].compareTo("" + 3062)==0) {//this
is the cap thread
			//port, requestID, NodeID, Operating
System
			HandleCreationCap(line);
		}
		else if(line[0].compareTo("" + 3052)==0) {//this
is the health thread
```

```
                //port, RequestID, NodeID, NodeHealth,
AttackType, VulnerabilityName, VulnerabilitySignature
                HandleCreationHealth(line);
           }
           else if(line[0].compareTo("" + 3042)==0) {//this
is the edge thread (HANDLES ONLY ONE EDGE AT A TIME)
                //port, requestID, NodeID, Edge Endpoint,
Throughput
                HandleCreationEdge(line);

           }
           else if(line[0].compareTo("" + 3032)==0) {//this
is the battery thread
                //port, requestID, NodeID, Battery Total,
Battery Power, Computation Rate,  Transfer Rate, Recieve
Rate
                HandleCreationBattery(line);
           }
           //should have made it through, so if not
somthing is wrong
           else {
                System.out.println("There was an error in
creating the node from the input");
           }
      }
      /////////////////////////////////////////////////////
/////////////////////////////
      /*
Function:    HandleCreationGeo
Arguments:   String [] line
Explanation: Takes in the array of string arguments
received over the network,
and creates the node with the given infotmation. Default
values are used when
none are specified.
Returns:
*/
      /////////////////////////////////////////////////////
/////////////////////////////
      private static void HandleCreationGeo (String []
line) {
           if(line.length<5) {
                System.out.println("In the creation
handler (port 3072) size " + line.length + " given and size
5 needed");
           }
           else {
                Vertex created = new
Vertex(Integer.parseInt(line[2]),
                           Double.parseDouble(line[3]),

      Double.parseDouble(line[4]));//geo contructor
```

```
                Graph.add(created);
            }
        }
        /////////////////////////////////////////////////////
/////////////////////////////
        /*
Function:    HandleCreationCap
Arguments:   String [] line
Explanation: Takes in the array of string arguments
received over the network,
and creates the node with the given infotmation. Default
values are used when
none are specified.
Returns:
*/
        /////////////////////////////////////////////////////
/////////////////////////////
        private static void HandleCreationCap (String []
line) {
            if(line.length<4) {
                System.out.println("In the creation
handler (port 3062) size " + line.length + " given and size
4 needed");
            }
            else {
                Vertex created = new
Vertex(Integer.parseInt(line[2]));//geo contructor
                Graph.add(created);
            }
        }
        /////////////////////////////////////////////////////
/////////////////////////////
        /*
Function:    HandleCreationHealth
Arguments:   String [] line
Explanation: Takes in the array of string arguments
received over the network,
and creates the node with the given infotmation. Default
values are used when
none are specified.
Returns:
*/
        /////////////////////////////////////////////////////
/////////////////////////////
        private static void HandleCreationHealth (String []
line) {
            if(line.length<6) {
                System.out.println("In the creation
handler (port 3052) size " + line.length + " given and size
7 needed");
            }
            else {
                int temp=0;
                temp=ParseHealth(line[3]);//converts
string to an integer
                Vertex created = new
Vertex(Integer.parseInt(line[2]),temp,line[4],
line[5]);//health contructor
                Graph.add(created);
           }
      }
      ///////////////////////////////////////////////////////
/////////////////////////////
      /*
Function:    HandleCreationEdge
Arguments:   String [] line
Explanation: Takes in the array of string arguments
received over the network,
and creates the node with the given infotmation. Default
values are used when
none are specified. If the edge endpoint doesnt exist it is
created.
Returns:
*/
      ///////////////////////////////////////////////////////
/////////////////////////////
      private static void HandleCreationEdge (String []
line) {
           if(line.length<5) {
                System.out.println("In the creation
handler (port 3042) size " + line.length + " given and size
5 needed");
           }
           else {
                Vertex created = new
Vertex(Integer.parseInt(line[2]));//geo contructor

                int batteryCost =
created.batteryTransferRate;//passes the normal batteryCost
                //must search through the graph for the
correct node, else create it
                int endLocation =
SearchForNode(Integer.parseInt(line[3]));

                if(endLocation!=-1) {//is in the graph
(the edge should never exist at this point, since origin
doesnt exist)
                     Edge tempEdge = new
Edge(Graph.get(endLocation),Integer.parseInt(line[4].trim()
),batteryCost);

      created.adjacencies.add(tempEdge);//adds the edge to
the vertex being created
                }
                else {//is in the graph (the edge should
never exist at this point, since origin doesnt exist)
                     Vertex tempNode = new
Vertex(Integer.parseInt(line[3]));
                     Graph.add(tempNode);
                     endLocation =
SearchForNode(Integer.parseInt(line[3]));
                     System.out.println("\n*********** "
+ Integer.parseInt(line[4].trim()));
                     Edge tempEdge = new
Edge(tempNode,Integer.parseInt(line[4].trim()),batteryCost)
;

     created.adjacencies.add(tempEdge);//adds the edge to
the vertex being created
                }
                Graph.add(created);
                //
if(Graph.get(endLocation).nodeID == created.nodeID){
                //              System.out.println("line "
+ line[0] + " " + line[1] + " " + line[2] + " " + line[3] +
" " + line[4]);
                //            }
                //now add the inverse edge, since the
origin + edge has now been created
                //            int firstLocation =
SearchForNode(created.nodeID);
                //            Edge inverseEdge = new
Edge(Graph.get(firstLocation),Integer.parseInt(line[4].trim
()),batteryCost);
                //
Graph.get(endLocation).adjacencies.add(inverseEdge);//adds
the inverse edge to the vertex
           }
     }
     ///////////////////////////////////////////////////////
/////////////////////////////
     /*
Function:    HandleCreationBattery
Arguments:   String [] line
Explanation: Takes in the array of string arguments
received over the network,
and creates the node with the given infotmation. Default
values are used when
none are specified.
Returns:
*/
     ///////////////////////////////////////////////////////
/////////////////////////////
     private static void HandleCreationBattery (String []
line) {
			//port, requestID, NodeID, Battery Total(j),
Battery Power, Computation Rate,  Transfer Rate, Recieve
Rate
			if(line.length<8) {
				System.out.println("In the creation
handler (port 3032) size " + line.length + " given and size
8 needed");
			}
			else {
				Vertex created = new
Vertex(Integer.parseInt(line[2]),
							Integer.parseInt(line[3]),
							Integer.parseInt(line[5]),
							Integer.parseInt(line[6]),

		Integer.parseInt(line[7].trim()));//geo contructor
				Graph.add(created);
			}
		}
		/////////////////////////////////////////////////////
/////////////////////////////
		/*
Function:	HandleUpdate
Arguments:	String [] line
Explanation: Takes in the array of string arguments
received over the network,
and depending on the port it was received on, updates the
node
with the given infotmation. If edge data is received, it
will
create end node if it doesn't exist.
Returns:
*/
		/////////////////////////////////////////////////////
/////////////////////////////
		private static void HandleUpdate (String [] line, int
graphLocation) {

			if(line[0].compareTo("" + 3072)==0) {//this is
the geo thread
				//port, requestID, NodeID, Lat, Long
				HandleUpdateGeo(line, graphLocation);
			}
			else if(line[0].compareTo("" + 3062)==0) {//this
is the cap thread
				//port, requestID, NodeID, Operating
System
				HandleUpdateCap(line, graphLocation);

			}
			else if(line[0].compareTo("" + 3052)==0) {//this
is the health thread
```

```
                //port, RequestID, NodeID, NodeHealth,
AttackType, VulnerabilityName, VulnerabilitySignature
                HandleUpdateHealth(line, graphLocation);

          }
          else if(line[0].compareTo("" + 3042)==0) {//this
is the edge thread
                HandleUpdateEdge(line,graphLocation);
          }
          else if(line[0].compareTo("" + 3032)==0) {//this
is the battery thread
                //port, requestID, NodeID, Battery Total,
Battery Power, Computation Rate, Transfer Rate, Recieve
Rate
                HandleUpdateBattery(line,graphLocation);

          }
          //should have made it through, so if not
somthing is wrong
          else {
                System.out.println("There was an error in
handling the given input");
          }

     }
     /////////////////////////////////////////////////////
/////////////////////////////
     /*
Function:    HandleUpdateGeo
Arguments:   String [] line
Explanation: Takes in the array of string arguments
received over the network,
and updates the node with the given geo information.
Returns:
*/
     /////////////////////////////////////////////////////
/////////////////////////////
     private static void HandleUpdateGeo (String [] line,
int graphLocation) {
          if(line.length<5) {
                System.out.println("In the update handler
(port 3072) size " + line.length + " given and size 5
needed");
          }
          else {
                Graph.get(graphLocation).latitude =
Double.parseDouble(line[3]);
                Graph.get(graphLocation).longitude =
Double.parseDouble(line[3]);
          }
     }
```

```
      ////////////////////////////////////////////////////
/////////////////////////////
      /*
Function:    HandleUpdateCap
Arguments:   String [] line
Explanation: Takes in the array of string arguments 
received over the network,
and updates the node with the given capability information.
Returns:
*/
      ////////////////////////////////////////////////////
/////////////////////////////
      private static void HandleUpdateCap (String [] line, 
int graphLocation) {
            if(line.length<5) {
                  System.out.println("In the update handler 
(port 3062) size " + line.length + " given and size 4 
needed");
            }
            //do nothing since os is unnecessary
      }
      ////////////////////////////////////////////////////
/////////////////////////////
      /*
Function:    HandleUpdateHealth
Arguments:   String [] line
Explanation: Takes in the array of string arguments 
received over the network,
and updates the node with the given health information.
Returns:
*/
      ////////////////////////////////////////////////////
/////////////////////////////
      private static void HandleUpdateHealth (String [] 
line, int graphLocation) {
            if(line.length<6) {
                  System.out.println("In the update handler 
(port 3052) size " + line.length + " given and size 7 
needed");
            }
            else {
                  Graph.get(graphLocation).health = 
ParseHealth(line[3]);//converts string to an integer
                  Graph.get(graphLocation).vulnerabilityName 
= line[4];

      Graph.get(graphLocation).vulnerabilitySignature = 
line[5];
            }
      }
      ////////////////////////////////////////////////////
/////////////////////////////
```

```
      /*
Function:    HandleUpdateEdge
Arguments:   String [] line
Explanation: Takes in the array of string arguments
received over the network,
and updates the node with the given edge information. If
the edge does not exist
it is created.
Returns:
*/
      ////////////////////////////////////////////////////
/////////////////////////////
      private static void HandleUpdateEdge (String [] line,
int firstLocation) {
            //port, requestID, NodeID, Edge Endpoint,
Throughput
            if(line.length<5) {
                  System.out.println("In the update handler
(port 3042) size " + line.length + " given and size 5
needed");
            }
            else {
                  int batteryCost =
Graph.get(firstLocation).batteryTransferRate;
                  //must search through the graph for the
correct node, else create it
                  int endLocation =
SearchForNode(Integer.parseInt(line[3]));
                  if(endLocation!=-1) {//is in the graph
                        //does the edge already exist?
                        Vertex firstVertex =
Graph.get(firstLocation);//gets the two vertexes for use in
the search for edge function
                        int secondID =
Integer.parseInt(line[3]);
                        int secondLocation =
SearchForNode(secondID);

                        Vertex secondVertex =
Graph.get(secondLocation);

                        int edgeLocation =
SearchForEdge(firstVertex,secondVertex);

                        if(edgeLocation==-1) {//edge doesn't
exist in the graph

      batteryCost+=Graph.get(endLocation).batteryReceiveRat
e;
                              Edge tempEdge = new
Edge(Graph.get(secondLocation),Integer.parseInt(line[4].tri
m()),batteryCost);
```

```
                            //Edge inverseEdge = new
Edge(Graph.get(firstLocation),Integer.parseInt(line[4].trim
()),batteryCost);

        Graph.get(firstLocation).adjacencies.add(tempEdge);//
adds the new edge to the vertex

        //Graph.get(secondLocation).adjacencies.add(inverseEd
ge);//adds the inverse edge to the vertex
                      }
                      else {//edge does exist in the graph
                            int updatedThroughput =
Integer.parseInt(line[4].trim());

        Graph.get(firstLocation).adjacencies.get(edgeLocation
).throughput=updatedThroughput;
                      }
                }
                else {
                      //if the end location is not in the
graph, then the edge does not exist
                      Vertex tempNode = new
Vertex(Integer.parseInt(line[3]));
                      Edge tempEdge = new
Edge(tempNode,Integer.parseInt(line[4].trim()),batteryCost)
;
                      //Edge inverseEdge = new
Edge(Graph.get(firstLocation),Integer.parseInt(line[4].trim
()),batteryCost);

        Graph.get(firstLocation).adjacencies.add(tempEdge);//
adds the edge to the vertex being created

        //tempNode.adjacencies.add(inverseEdge);//adds the
inverse edge to the vertex
                      Graph.add(tempNode);
                }
          }
    }
    /////////////////////////////////////////////////////
/////////////////////////////
    /*
Function:    HandleUpdateBattery
Arguments:   String [] line
Explanation: Takes in the array of string arguments
received over the network,
and updates the node with the given battery information.
Returns:
*/
    /////////////////////////////////////////////////////
/////////////////////////////
```

```
     private static void HandleUpdateBattery (String []
line, int graphLocation) {
          if(line.length<7) {
               System.out.println("In the update handler
(port 3032) size " + line.length + " given and size 6
needed");
          }
          else {
               Graph.get(graphLocation).batteryRemaining
= Integer.parseInt(line[3]);

     Graph.get(graphLocation).batteryComputationRate =
Integer.parseInt(line[5]);

     Graph.get(graphLocation).batteryTransferRate =
Integer.parseInt(line[6]);

     Graph.get(graphLocation).batteryReceiveRate =
Integer.parseInt(line[7].trim());
          }
     }
     ////////////////////////////////////////////////////
/////////////////////////////
     /*
Function:    SearchForEdge
Arguments:   String health
Explanation: Searches for an edge on the given node which
points to the end
location.
Returns:     Returns the index in the edge array searched
for. Returns -1 if
not found.
*/
     ////////////////////////////////////////////////////
/////////////////////////////
     private static int SearchForEdge(Vertex start, Vertex
end) {
          ArrayList<Edge> edges = start.adjacencies;
          for(int ind=0; ind<edges.size(); ind++) {
               Edge temp = edges.get(ind);
               if(temp.target.nodeID==end.nodeID) {//if
node ID's are the same
                    return ind;
               }
          }
          return -1;
     }
     ////////////////////////////////////////////////////
/////////////////////////////
     /*
Function:   ParseHealth
Arguments:   String health
```

```
Explanation: Takes in the health as a string and converts
it to a integer type.
Returns:     Returns the health of the node as an integer
*/
     ////////////////////////////////////////////////////
/////////////////////////////
     private static int ParseHealth(String health) {
          if(health.compareTo("infected")==0) {
               return 1;
          }
          else if(health.compareTo("vulnerable")==0) {
               return 2;
          }
          else if(health.compareTo("suseptible")==0) {
               return 3;
          }
          else if(health.compareTo("immune")==0) {
               return 4;
          }
          else {
               System.out.println("Node health not parsed
correctly: " + health + " returning vulnerable.");
               return 2;
          }
     }
     ////////////////////////////////////////////////////
/////////////////////////////
     /*
Function:    SearchForNode
Arguments:   int nodeID
Explanation: Takes in the nodeID and searches the graph for
the node. Returns the
location of the node requested. Returns -1 if not found.
Returns:     Returns the location of the node in the graph
array.
*/
     ////////////////////////////////////////////////////
/////////////////////////////
     private static int SearchForNode(int nodeID) {
          for(int ind=0; ind<Graph.size(); ind++) {
               if(Graph.get(ind).nodeID == nodeID) {
                    return ind;
               }
          }
          return -1;
     }
     ////////////////////////////////////////////////////
/////////////////////////////
     /*
Function:    SearchForImmuneNodes
Arguments:
Explanation: Searches the graph for immune nodes.
```

```
Returns:     Returns a list of the immune vertexes in the
graph
*/
     ///////////////////////////////////////////////////////
//////////////////////////////
     private static ArrayList<Vertex>
SearchForImmuneNodes() {
          ArrayList<Vertex> temp = new
ArrayList<Vertex>();
          for(int ind=0; ind<Graph.size(); ind++) {
               if(Graph.get(ind).health == 4) {//if the
node is immune/patched
                    temp.add(Graph.get(ind));
               }
          }
          return temp;
     }
     ///////////////////////////////////////////////////////
//////////////////////////////(shouldnt need to use this)
     /*
Function:    UpdateSuseptible
Arguments:
Explanation: Loops through all of the graph edges and
update the correct nodes to
be suseptible.
Returns:
*/
     ///////////////////////////////////////////////////////
//////////////////////////////
     private static void UpdateSuseptible() {
          for(int ind=0; ind<Graph.size(); ind++) {
               Vertex node =Graph.get(ind);
               if(node.health==1) {
                    for( int ind2=0;
ind2<node.adjacencies.size(); ind2++) {
                         Edge next =
node.adjacencies.get(ind2);
                         next.target.health=3;
                    }
               }
          }
     }
     ///////////////////////////////////////////////////////
//////////////////////////////
     /*
Function:    UpdateEdges
Arguments:
Explanation: Loops through all of the graph edges and
update the edge weights with
the correct amounts.
Returns:
*/
```

```
      ///////////////////////////////////////////////////////
/////////////////////////////
      private static void UpdateEdges() {
            for(int ind=0; ind<Graph.size(); ind++) {
                  Vertex node =Graph.get(ind);
                  for( int ind2=0;
ind2<node.adjacencies.size(); ind2++) {
                        Edge next =
node.adjacencies.get(ind2);
                        int temp = node.batteryTransferRate;
                        temp+=next.target.batteryReceiveRate;
                        next.batteryCost=temp;
                  }
            }
      }
      ///////////////////////////////////////////////////////
/////////////////////////////
      /*
Function:    UpdateCriticality
Arguments:   ArrayList<Vertex> criticalPath
Explanation: Loops through all of the vertexes in the given
list makes them
critical in the graph
Returns:
*/
      ///////////////////////////////////////////////////////
/////////////////////////////
      private static void
UpdateCriticality(ArrayList<Vertex> criticalPath) {
            for(int ind=0; ind<criticalPath.size(); ind++) {
                  Vertex next = criticalPath.get(ind);
                  int location = SearchForNode(next.nodeID);
                  Graph.get(location).critical=true;
            }
      }
      ///////////////////////////////////////////////////////
/////////////////////////////
      /*
Function:    SendPath
Arguments:   int requestID, ArrayList<Vertex> path, int
gui, int cyfi,
int cost
Explanation: Sends the path to the on the two given ports,
with the given
cost.
Returns:
*/
      ///////////////////////////////////////////////////////
/////////////////////////////
      private static void SendPath(int requestID,
ArrayList<Vertex> path, int gui, int cyfi, double cost)
{//3081 to chuck
```

```
            String tupleMessage = "";
            System.out.println("Sending path data " + 
path.size());
            for(int ind=0; ind<path.size()-1; ind++) {
                  tupleMessage =requestID + "," + 
path.get(ind).nodeID + "," + path.get(ind+1).nodeID  + "," 
+ cost + ",";
                  System.out.println("PATH MESSAGE " + 
tupleMessage);//shows what is outputted to Chuck
                  try {
                        DatagramSocket outputThreadSocket = 
new DatagramSocket();//output path socket

      outputThreadSocket.setSoTimeout(15000);
                        byte[] sendThreadLine = new 
byte[1024];

      System.arraycopy(tupleMessage.getBytes(),0,sendThread
Line,0,tupleMessage.length());
                        //begins sending the message to GUI
                        System.out.println("localAddress " + 
localaddr);
                        DatagramPacket sendGUIPacket = new 
DatagramPacket(sendThreadLine, sendThreadLine.length, 
localaddr, gui);
                        //begins sending the same message to 
CyFi
                        DatagramPacket sendCyfiPacket = new 
DatagramPacket(sendThreadLine, sendThreadLine.length, 
localaddr, cyfi);

      outputThreadSocket.send(sendGUIPacket);

      outputThreadSocket.send(sendCyfiPacket);
                        if(ind == path.size() - 2){
                              String tupleMessageFinal = 
requestID + "," + path.get(ind + 1).nodeID + "," + -1  + 
"," + cost + ",";
                              DatagramPacket 
sendCyfiPacketFinal = new 
DatagramPacket(tupleMessageFinal.getBytes(), 
tupleMessageFinal.getBytes().length, localaddr, cyfi);

      outputThreadSocket.send(sendCyfiPacketFinal);
                        }
                        if(ind==(path.size()-2)) {
                              outputThreadSocket.close();
                        }
                  }
                  catch (SocketTimeoutException e) {
                        System.out.println("output to a 
sendpath socket has timed out");
```

```
                }
                catch (PortUnreachableException e) {
                     System.out.println("output to a
sendpath: port chosen was unreachable");
                }
                catch (IOException e) {
                     System.out.println("an IO error has
occurred from send or recieve in output on sendpath ");
                }
                catch  (Exception e) {
                     System.out.println("network code
failed for an unknown reason in output on sendpath ");
                }
           }
      }
      ////////////////////////////////////////////////////
/////////////////////////////
      /*
Function:    SendCost
Arguments:   int requestID, int cyfi, int cost
Explanation: Sends the cost to the given hostname on the
given port.
Returns:
*/
      ////////////////////////////////////////////////////
/////////////////////////////
      private static void SendCost(int requestID, int cyfi,
double cost) {//3081 to chuck
           String tupleMessage = "";
           tupleMessage =requestID + "," + cost + ",";

           System.out.println(tupleMessage +
"*****Message");//shows what is outputted to Chuck
           try {
                DatagramSocket outputThreadSocket = new
DatagramSocket();//output path/cost socket
                outputThreadSocket.setSoTimeout(15000);
                byte[] sendThreadLine = new byte[1024];

      System.arraycopy(tupleMessage.getBytes(),0,sendThread
Line,0,tupleMessage.length());
                DatagramPacket sendCyfiPacket = new
DatagramPacket(sendThreadLine, sendThreadLine.length,
localaddr, cyfi);
                outputThreadSocket.send(sendCyfiPacket);
                outputThreadSocket.close();

           }
           catch (SocketTimeoutException e) {
                System.out.println("output to a cyfi
socket has timed out (cost)");
           }
```

```
            catch (PortUnreachableException e) {
                  System.out.println("output to a sendpath:
port chosen was unreachable (cost)");
            }
            catch (IOException e) {
                  System.out.println("an IO error has
occurred from send or recieve in output on send cost ");
            }
            catch  (Exception e) {
                  System.out.println("network code failed
for an unknown reason in output on send cost ");
            }
      }
      ////////////////////////////////////////////////////
/////////////////////////////
      /*
Function:    HandleRequest
Arguments:   String [] line
Explanation: Takes in the array of string arguments
received over the network,
and handles the request, sending back the correct requested
info.
Returns:
*/
      ////////////////////////////////////////////////////
/////////////////////////////
      private static void HandleRequest(String [] line) {
            //port, requestID,startNode, endNode, plan #,
patchsize (should be 0 in all plans but "4")

            UpdateEdges();//must update edge weights !! or
they are wrong
            int requestID = Integer.parseInt(line[1]);
            int startingID = Integer.parseInt(line[2]);
            int endingID = Integer.parseInt(line[3]);
            int operation = Integer.parseInt(line[4]);
            double sizeD = Double.parseDouble(line[5]);
            int size = (int)sizeD;
            double cost = 0.0;

            System.out.println("request  operation " +
operation);

            //**************** patch cost (sends patch plan)
            System.out.println("HANDLE PATCH");
            int startLocation = SearchForNode(startingID);
            int endLocation = SearchForNode(endingID);

            System.out.println("\n Secure Path");
            ArrayList<Vertex> secureThroughputPath =
SecurePath(startLocation, endLocation, true);
```

```
            double
throughputCost=ComputeCost(secureThroughputPath,true,size);

      SendPath(requestID,secureThroughputPath,9999,3081,thr
oughputCost);

            System.out.println("\n Send Path");
            ArrayList<Vertex> secureBatteryPath =
SecurePath(startLocation, endLocation, false);
            double
batteryCost=ComputeCost(secureBatteryPath,false,size);

      SendPath(requestID,secureBatteryPath,9999,3081,batter
yCost);

            cost = throughputCost + batteryCost; //this is
the cost used to compute other costs
            System.out.println("\ncost: " + cost + "
throughput: " + throughputCost + " battery: " +
batteryCost);

            //*****************IP Block
            System.out.println("\n IP Block");
            SendCost(requestID,3081, 0);


            //****************Wall off
            System.out.println("\n Wall Off");
            Vertex tempNode = Graph.get(startLocation);
            tempNode.state=2;
            SendCost(requestID,3081, cost * .25);


            //****************Heal
            System.out.println("\n Heal");
            System.out.println("Sending heal request ");
            Vertex tempNode2 = Graph.get(startLocation);
            tempNode2.state = 3;
            SendCost(requestID,3081, cost * .75);

            //Patch Critical path (critical path plan)
            System.out.println("\n Critical Path");
            ArrayList<Vertex> criticalPath =
SecurePath(startLocation, endLocation, true);

      SendPath(requestID,criticalPath,4001,3081,0);//cost
is 0 because it is a critical path


            System.out.println("Error, operation- requested
is not found: " + operation);
```

```
     }

     ////////////////////////////////////////////////////
/////////////////////////////
     /*
Function:    HandleGuiPath
Arguments:   String [] line
Explanation: Takes in the array of string arguments
received over the network,
and handles the request, sending back the critical path
route over the network.
Returns:
*/
     ////////////////////////////////////////////////////
/////////////////////////////
     private static void HandleGuiPath(String [] line) {
          //port, reqestID, origin, endpoint
          System.out.println("Critical Path");
          int requestID = Integer.parseInt(line[1]);
          UpdateEdges();//must update edge weights !! or
they are wrong
          System.out.println("Received starting critical
path request, printing graph");
          PrintGraph();

          System.out.println("start " +
Integer.parseInt(line[2].trim()) + " end " +
Integer.parseInt(line[3].trim()));
          ArrayList<Vertex> criticalPath=
SecurePath(Integer.parseInt(line[2].trim()),Integer.parseIn
t(line[3].trim()) , true);
          UpdateCriticality(criticalPath);
          SendPath(requestID,criticalPath,4001,3081,0);
     }
     /*
Function:    ResetGraphForPathing
Arguments:
Explanation: Iterates through the array and resets the
minDistances so a new
route can be calculated.
Returns:
*/
     private static void ResetGraphForPathing() {
          for(int ind=0; ind<Graph.size(); ind++) {

     Graph.get(ind).minDistance=Double.POSITIVE_INFINITY;
          }
     }
     ////////////////////////////////////////////////////
/////////////////////////////
     /*
Function:    CriticalPath
```

```
Arguments:   int origin, int endpoint
Explanation: Takes in the origin and endpoint, and returns
the best critical path
possible in the form of an Array List. It does not consider
infected nodes.
Returns: Returns the critical path in the form of an
ArrayList<Vertex>. The
ArrayList is empty if no path is possible.
*/
	////////////////////////////////////////////////////
/////////////////////////////
	private static ArrayList<Vertex> SecurePath(int
origin, int endpoint, boolean isThroughput) {
		UpdateEdges();//must update edge weights !! or
they are wrong
		//UpdateSuseptible();
		ArrayList<Vertex> immune =
SearchForImmuneNodes();//must make a search for infected
nodes

		System.out.println("Received a Secure Route
request, printing graph");
		PrintGraph();

		ArrayList<Vertex> clean = new
ArrayList<Vertex>();
		for(int ind=0; ind<Graph.size(); ind++) {
			Vertex temp = Graph.get(ind);
			if(temp.health!=1) {//if the node is not
infected
				clean.add(temp);
			}
		}
		//should have a graph with no infected nodes
		int originLocation = -1;
		for(int ind=0; ind<clean.size(); ind++) {
			Vertex temp = clean.get(ind);
			if(temp.nodeID==origin) {//if the node is
the origin
				originLocation=ind;//sets the
location to the one found
			}
		}
		int endLocation = -1;
		for(int ind=0; ind<clean.size(); ind++) {
			Vertex temp = clean.get(ind);
			if(temp.nodeID==endpoint) {//if the node
is the endpoint
				endLocation=ind;//sets the location
to the one found
			}
		}
		if(originLocation==-1) {//origin node not found
			System.out.println("Secure Path: Origin
node not found in clean list ");
			int overallOriginLocation =
SearchForNode(origin);

	if(Graph.get(overallOriginLocation).health==1)
{//node is infected
				System.out.println("origin node
infected");
			}
		}
		else if(endLocation==-1) {//endpoint not found
			System.out.println("Secure Path: End node
not found in clean list ");
			int overallEndLocation =
SearchForNode(endpoint);

	if(Graph.get(overallEndLocation).health==1) {//node
is infected
				System.out.println("End node
infected");
			}
		}
		else {
			//clear any previous run of secure path
			for(Vertex v : clean){
				v.previous = null;
				v.minDistance =
Double.POSITIVE_INFINITY;
			}

			if(isThroughput) {//if throughput was
selected

	ComputePaths(clean.get(originLocation), true);
				ArrayList<Vertex> throughputPath =
GetShortestPathTo(clean.get(endLocation));
				return throughputPath;
			}
			else {

	ComputePaths(clean.get(originLocation), false);
				ArrayList<Vertex> batteryPath =
GetShortestPathTo(clean.get(endLocation));
				//ResetGraphForPathing();//only
necessary if changes werent just made on clean
				return batteryPath;
			}
		}
		ArrayList<Vertex> empty = new
ArrayList<Vertex>();
```

```
           return empty;

      }
      //////////////////////////////////////////////////////
/////////////////////////////
      /*
Function:     PrintGraph
Arguments:
Explanation: Linearly moves through the graph and runs the
toString method on
each vertex in the graph.
Returns:
*/
      //////////////////////////////////////////////////////
/////////////////////////////
      private static void PrintGraph() {
           /*for(int ind=0; ind<Graph.size(); ind++) {
             System.out.println(Graph.get(ind).toString());
             }*/
      }
      //////////////////////////////////////////////////////
/////////////////////////////
      /*
Function:     TestOutput
Arguments:
Explanation: Creates and sends a message back to Cra.local
on a given port. Used
solely for testing nework receive code.
Returns:
*/
      //////////////////////////////////////////////////////
/////////////////////////////
      private static void TestOutput(int port, String
message) {
           try {
                 DatagramSocket outputThreadSocket = new
DatagramSocket(3022);//output socket
                 outputThreadSocket.setSoTimeout(15000);
                 byte[] sendThreadLine = new byte[1024];
                 String testThreadMessage = message;

      System.arraycopy(testThreadMessage.getBytes(),0,sendT
hreadLine,0,testThreadMessage.length());

                 //InetAddress outgoingThreadAddress =
InetAddress.getByName("CRA.local");
                 //eventually this next line will be sent
to port 3020
                 DatagramPacket sendPacket = new
DatagramPacket(sendThreadLine, sendThreadLine.length,
localaddr, port);
                 while(!outputThreadSocket.isClosed()) {
                        outputThreadSocket.send(sendPacket);

        outputThreadSocket.close();/////closes because i am
only sending one thing
                    }
            }
            catch (SocketTimeoutException e) {
                    System.out.println("a socket has timed
out");
            }
            catch (PortUnreachableException e) {
                    System.out.println("a port chosen was
unreachable");
            }
            catch (IOException e) {
                    System.out.println("an IO error has
occurred from send or recieve");
            }
            catch  (Exception e) {
                    System.out.println("network code failed
for an unknown reason");
            }
        }
        ////////////////////////////////////////////////////
/////////////////////////////
        /*
Function:    ComputeCost
Arguments:   ArrayList<Vertex> path, Boolean isThroughput
Explanation: Takes in the best path route and computes the
cost on either
battery and throughput.
Returns:     Returns the cost of the shortest path along
either throughput or
battery lines.
*/
        ////////////////////////////////////////////////////
/////////////////////////////
        private static double ComputeCost(ArrayList<Vertex>
path, boolean isThroughput, int patchSize) {
            double sum = 0.0;
            double cost = 0.0;
            for(int ind=0; ind<(path.size()-1); ind++)
{//only needs to go to the second to last point since edge
is used
                    Vertex temp = path.get(ind);
                    //search through the edges until you get
the next vertex path[ind] is next vertex
                    int edgeToUse=-1;
                    for(int e=0; e<temp.adjacencies.size();
e++) {
      if(temp.adjacencies.get(e).target.nodeID ==
path.get(ind+1).nodeID) {//endpoint is next vertex
                           edgeToUse=e;

      e=temp.adjacencies.size();//ends the loop since the
the next vertex has been found
                      }
                 }
                 if(edgeToUse==-1) {
                      //System.out.println("there was an
error in cost computation, next edge not found");
                      return -1;
                 }
                 //edge to use should have been changed at
this point
                 if(isThroughput) {

      sum+=temp.adjacencies.get(edgeToUse).throughput;//edg
eToUse is out of bounds
                 }
                 else {
                      sum +=
temp.adjacencies.get(edgeToUse).batteryCost;//battery
transfer cost + recieve cost Whrs
                 }

            }
            if(isThroughput) {
                 cost = (double)(sum *
patchSize);//edgeToUse is out of bounds
            }
            else {
                 //at this point, sum is total of the
network in Whrs
                 //we want the cost in (joules sum / 3600)
* time to completion
                 cost = (double)(patchSize*(sum/3600));
            }
            return cost;
      }
      ///////////////////////////////////////////////////
/////////////////////////////
      /*
Function:    ComputeCompletionTime
Arguments:   ArrayList<Vertex> path, int patchSize
Explanation: Takes in the best path route and computes the
time to complete the
patch operation given the patchSize and throughput from
each edge.
Returns:     Returns the completion time of the shortest
path.
```

```
*/
      /////////////////////////////////////////////////////
/////////////////////////////
      private static double
ComputeCompletionTime(ArrayList<Vertex> path, int
patchSize) {
            double time=0;
            for(int ind=0; ind<(path.size()-1); ind++) {
                  Vertex temp = path.get(ind);
                  //search through the edges until you get
the next vertex path[ind] is next vertex
                  int edgeToUse=0;
                  for(int e=0; e<temp.adjacencies.size();
e++) {

      if(temp.adjacencies.get(e).target.nodeID ==
path.get(ind+1).nodeID) {//if edge endpoint is next
                              edgeToUse=e;

      e=temp.adjacencies.size();//ends the loop since the
the next vertex has been found
                        }
                  }
                  //edge to use should have been changed at
this point
                  if(edgeToUse==-1) {
                        System.out.println("there was an
error in time calculation, next edge not found");
                        return -1;
                  }

      time+=patchSize*temp.adjacencies.get(edgeToUse).throu
ghput;
            }
            return time;
      }
      /////////////////////////////////////////////////////
/////////////////////////////
      /*
Class:       ListenThread
Explanation: This class is a listener on a given port which
pushes received data
onto the blocking queue when it is received.
*/
      /////////////////////////////////////////////////////
/////////////////////////////
      private static class ListenThread implements
Runnable{
            public int listenPort;
            /*
Method:      ListenThread basic contructor
Arguments:   int port
```

```
Explanation: Takes the given port information and sets the
class variable
to that port.
*/
           public ListenThread(int port) {
                listenPort = port;
           }
           /*
Method:      Run
Arguments:
Explanation: Listens on the chosen port and when data is
received, it
pushes the data onto the BlockingQueue.
*/
           public void  run() {
                try {
                     DatagramSocket listenSocket = new
DatagramSocket(listenPort);
                     //listenSocket.setSoTimeout(15000);
                     byte[] receivedData = new byte[1024];
                     DatagramPacket receivedPacket = new
DatagramPacket(receivedData, receivedData.length);
                     while(true) {
                          //receives packet

     listenSocket.receive(receivedPacket); //(should
overwrite the last packet sent)
                          //(testing)output what was
given
                          String inData = new
String(receivedPacket.getData());
                          inData = listenPort + "," +
inData;
                          //String port =
                          //push to the queue of work
availiable
                          synchronized (this) {
                               workToDo.put(inData);

     //System.out.println("added this to queue: " +
inData);
                          }
                     }
                }
                catch (SocketTimeoutException e) {
                     System.out.println("a socket has
timed out on port: " + listenPort);
                }
                catch (PortUnreachableException e) {
                     System.out.println("a port chosen was
unreachable on port: " + listenPort);
                }
                catch (IOException e) {
                     e.printStackTrace();
                     //System.out.println("an IO error has 
occurred from send or recieve on port: " + listenPort);
                }
                catch  (Exception e) {
                     e.printStackTrace();
                     //System.out.println("network code 
failed for an unknown reason on port: "  + listenPort);
                }
           }
      }
      /////////////////////////////////////////////////////
/////////////////////////////
      /*
Function:    ComputePaths
Arguments:   Vertex Source, boolean isThroughput
Explanation: takes in the source route and 
patch operation given the patchSize and throughput from 
each edge.
Returns:     Returns the completion time of the shortest 
path.
*/
      /////////////////////////////////////////////////////
/////////////////////////////
      public static void ComputePaths(Vertex source, 
boolean isThroughput) {//yes = throughput ,  no = 
batteryCost
           source.minDistance = 0.;
           PriorityQueue<Vertex> vertexQueue = new 
PriorityQueue<Vertex>();
           vertexQueue.add(source);

           while (!vertexQueue.isEmpty()) {//goes until it 
is empty
                Vertex u = vertexQueue.poll();//gives the 
head of the queue

                // Visit each edge exiting u
                for (Edge e : u.adjacencies) {
                     Vertex v = e.target;//gets the end 
vertex
                     double weight;
                     if(isThroughput) {
                          weight= e.throughput;
                     }
                     else {
                          weight = e.batteryCost;
                     }
                     double distanceThroughU = 
u.minDistance + weight;//adds weight to the shortest route 
so far
```

```
                    if (distanceThroughU < v.minDistance)
{//if the weight so far is less than other shortest
                         vertexQueue.remove(v);
                         v.minDistance =
distanceThroughU ;
                         v.previous = u;
                         vertexQueue.add(v);//keep
exploring by putting this vertex on the queue
                    }
               }
          }
     }
     ///////////////////////////////////////////////////////
/////////////////////////////
     /*
Function:    GetShortestPathTo
Arguments:   Vertex target
Explanation: Computes the shortest path to the chosen end
node using Dijkstra's
Algorithm.
Returns:     Returns the ArrayList<Vertex> of the shortest
path to the node.
*/
     ///////////////////////////////////////////////////////
/////////////////////////////
     public static ArrayList<Vertex>
GetShortestPathTo(Vertex target) {
          ArrayList<Vertex> path = new
ArrayList<Vertex>();
          for (Vertex vertex = target; vertex != null;
vertex = vertex.previous){
               path.add(vertex);
          }
          Collections.reverse(path);
          return path;
     }
     ////////////////////////////////////////Vertex and
Edge Classes////////////////////
     public static class Vertex implements
Comparable<Vertex> {
          // infected, vulnerable, immune - critical -
suseptible
          public final int nodeID;
          public ArrayList<Edge> adjacencies;
          public double minDistance =
Double.POSITIVE_INFINITY;
          public Vertex previous;
          public int batteryRemaining;//the amount of
battery left in the device(J)
          public int batteryTransferRate;//in Whrs
          public int batteryReceiveRate;//in Whrs
          public int batteryComputationRate;//in Whrs
          public String vulnerabilityName;
          public String vulnerabilitySignature;
          public double latitude;
          public double longitude;
          public boolean critical;//unused
          public int health;//infected,
vulnerable,suseptible,immune,
          public int state; //normal, healing, walling
(unused)

          /*
Method:      Edge basic contructor
Arguments:   int nodeID
Explanation: Takes the given information and creates the
vertex.
*/
          public Vertex(int nodeID) {//constructor if
given the node id only
               this.nodeID = nodeID;
               adjacencies=new ArrayList<Edge>();
               batteryRemaining=100;
               batteryTransferRate=1;
               batteryReceiveRate=1;
               batteryComputationRate=1;
               vulnerabilityName="";
               vulnerabilitySignature="";
               latitude=0;
               longitude=0;
               critical=false;
               health=2;
               state=1;
          }
          /*
Method:      Edge overloaded contructor
Arguments:   int nodeID, int Lat, int Long
Explanation: Takes the given information and creates the
vertex.
*/
          public Vertex(int nodeID, double lat, double
lon) {
               //port, requestID, NodeID, Lat, Long
               this.nodeID = nodeID;
               adjacencies=new ArrayList<Edge>();
               batteryRemaining=100;
               batteryTransferRate=1;
               batteryReceiveRate=1;
               batteryComputationRate=1;
               vulnerabilityName="";
               vulnerabilitySignature="";
               latitude=lat;
               longitude=lon;
               critical=false;
```

```
                health=2;
                state=1;
           }
           /*
Method:      Edge overloaded contructor
Arguments:   int nodeID, String name, String sig
Explanation: Takes the given information and creates the
vertex.
*/
           public Vertex(int nodeID, int health, String
name, String sig) {
                this.nodeID = nodeID;
                adjacencies=new ArrayList<Edge>();
                batteryRemaining=100;
                batteryTransferRate=1;
                batteryReceiveRate=1;
                batteryComputationRate=1;
                vulnerabilityName=name;
                vulnerabilitySignature=sig;
                latitude=0;
                longitude=0;
                critical=false;
                health=health;
                state=1;
           }
           /*
Method:      Edge overloaded contructor
Arguments:   int nodeID, int battery, int transfer, int
receive
Explanation: Takes the given information and creates the
vertex.
*/
           public Vertex(int nodeID, int battery, int
compute, int transfer, int receive) {
                this.nodeID = nodeID;
                adjacencies=new ArrayList<Edge>();
                batteryRemaining=battery;
                batteryTransferRate=transfer;
                batteryReceiveRate=receive;
                batteryComputationRate=compute;
                vulnerabilityName="";
                vulnerabilitySignature="";
                latitude=0;
                longitude=0;
                critical=false;
                health=2;
                state=1;
           }
           /*
Method:      toString
Arguments:
```

```
Explanation: Sends back a string representation of a
Vertex.
*/
           public String toString() {

                String name = nodeID + ": \n";
                String battery = "Battery Remaining: " +
batteryRemaining +
                     " Battery Compute Rate: " +
batteryComputationRate +
                     " Battery Transfer Rate: " +
batteryTransferRate +
                     " Battery Receive Rate: " +
batteryReceiveRate + "\n";

                String vuln = "Vulnerability Name: " +
vulnerabilityName +
                     " Vulnerability Signature: " +
vulnerabilitySignature + "\n";

                String gps = "Latitude: " + latitude +
                     " Longitude: " + longitude + "\n";

                String edges = "";

                for(int ind=0; ind<adjacencies.size();
ind++) {
                     edges+=" " +
adjacencies.get(ind).toString();
                }
                edges+="\n";
                String total = name + battery + vuln + gps
+ edges;
                return total;
           }
           /*
Method:      compareTo
Arguments:   Vertex orther
Explanation: sends back which min distance is shorter
*/
           public int compareTo(Vertex other) {
                return Double.compare(minDistance,
other.minDistance);
           }
      }
      ///////////////////////////////////////////////////
/////////////////////////////
      /*
Class:       Edge
Explanation: This class is a representation of the edges in
the graph. It
```

```
principally contains the end vertex and two different
weights:
throughput and battery cost.
*/
      /////////////////////////////////////////////////////
/////////////////////////////
      public static class Edge {
            public final Vertex target;
            public double throughput; //seconds/Mb
            public int batteryCost; //transfer of one +
recieve of other
            /*
Method:       Edge basic contructor
Arguments:    Vertex argTarget, int argThroughput, int
argBattery
Explanation: Takes the given edge information and creates
the edge.
*/
            public Edge(Vertex argTarget, int argThroughput,
int argBattery){
                  target = argTarget;
                  throughput = (1.0/argThroughput);
                  batteryCost = argBattery;
            }
            /*
Method:       toString
Arguments:
Explanation: Sends back a string representation of an Edge.
*/
            public String toString() {
                  return "target: " + target.nodeID + ", "+
throughput +", " + batteryCost;
            }
      }
}
```

List of Symbols, Abbreviations, and Acronyms

CyFiA	Cyber Fighter Associate
CRA	Collaborative Research Alliance
CSec	Cyber-Security
GUI	graphical user interface
OS	operating system
SIIS	Systems and Internet Infrastructure Security

1	DEFENSE TECHNICAL
(PDF)	INFORMATION CTR
	DTIC OCA
2	DIRECTOR
(PDF)	US ARMY RESEARCH LAB
	IMAL HRA
	RDRL CIO LL
1	GOVT PRINTG OFC
(PDF)	A MALHOTRA
2	DIR USARL
(PDF)	RDRL CIN D
	L MARVEL
	RDRL CIN T
	A SWAMI